THE EXPERT'S VOICE® IN .NET

Programming Reactive Extensions and LINQ

DIVE DEEP INTO THE NEXT IMPORTANT .NET TECHNOLOGY

Jesse Liberty and Paul Betts

Apress®

Programming Reactive Extensions and LINQ

Jesse Liberty
Paul Betts

Apress®

Programming Reactive Extensions and LINQ

ISBN-13 (pbk): 978-1-4302-3747-1

ISBN-13 (electronic): 978-1-4302-3748-8

President and Publisher: Paul Manning
Lead Editor: Ewan Buckingham
Technical Reviewer: Stefan Turalski
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Engel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Jessica Belanger
Copy Editor: Kimberly Burton
Indexer: BIM Indexing & Proofreading Services
Production Support: Patrick Cunningham
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

The source code for this book is available to readers at `www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

This book is dedicated to my mother, Edythe Levine, who will not understand one word of it, but will love it anyway.

—Jesse Liberty

This book is dedicated to my wife, Ulrike Stoll, who has been and continues to be infinitely understanding about my late nights spent hacking on the next big idea.

—Paul Betts

Contents at a Glance

Contents

About the Authors

Jesse Liberty is a senior developer-community evangelist on the Microsoft Windows Phone team. He hosts the popular "Yet Another Podcast" on his blog at `http://JesseLiberty.com`. The entire blog is required reading. Liberty is the co-author of numerous top-selling books, including *Migrating to Windows Phone 7* by (Apress, 2011) and *Programming C# 4.0* (O'Reilly Media, 2010). Prior to Microsoft, he was a distinguished software engineer at AT&T, a software architect for PBS, and vice president of information technology at Citibank. He can be followed on Twitter `@JesseLiberty`.

Paul Betts is a software developer and Hubbernaut at GitHub, where he works on bringing the joy of Git and GitHub to the world of .NET and Windows. Paul previously worked at Microsoft in the Windows and Office organizations. His blog can be found at `http://blog.paulbetts.org`. Follow him on Twitter at `@xpaulbettsx`.

About the Technical Reviewer

Stefan Turalski is a nice chap who is capable of performing both magical and trivial things with a little help from code, libraries, tools, APIs, servers, and the like. He is experienced in almost all aspects of the software lifecycle, and is especially skilled in business analysis, design, implementation, testing, and QA.

Turalski's areas of interest are wide, best summarized as emerging technologies. His current focus is on .NET 4.5 , GPGPU, event-driven architectures, and software engineering at large. Before he realized that he enjoys criticizing other people's work more, Stefan published several technical articles, mainly about .NET technology, SOA (service-oriented architecture), and software engineering.

For the last ten or so years, Turalski has built solutions ranging from Perl scripts, embedded systems, and web sites, to highly scalable C++/Java/.NET enterprise class systems. Feel free to contact him at `stefan.turalski@ gmail.com`.

Acknowledgments

Thank you to Paul Betts whose genius was the sine qua non of this entire project. Special thanks to John Osborn, Jessica Belanger, Stefan Turalski, Ewan Buckingham, Kimberly Burton and all the folks at Apress for bringing this book to life and making it a far better book than the one we originally wrote. And thank you, dear reader, without whom this book would be a paperweight.

—Jesse Liberty

Thanks to Erik Meijer, Bart de Smet, Wes Dyer, Matthew Podwysocki, and the rest of the Rx team. They have brought their brilliant ideas to life by creating the Reactive Extensions. Thanks to my co-author, Jesse Liberty, for explaining my far too terse samples in a language that human beings can actually understand. Thanks also to our technical editor and the great people at Apress for their hard work in making this a great book.

—Paul Betts

Foreword

For the past fifteen years, like a cyber-age captain Ahab, I have been hunting the unruly, fail whale of web programming. When you move from a single machine to the distributed world of the web, all of your existing assumptions about programming fail to hold. In particular, you suddenly have to face problems such as latency, network errors, and heterogeneity in data models, size, and velocity. You can try to ignore them, but they will invariably come back to bite off your leg.

Whenever I face a seemingly impossible problem like this, I turn to my secret weapon, mathematics, and in particular, category theory. Mathematicians often refer to category theory as "generalized abstract nonsense," which actually is a rather accurate moniker. Category theory is the mathematician's version of interface-based programming. As a result, the true essence of the problem surfaces, unmuddled by implementation details, and as a result, often uncovers unexpected connections between seemingly different concrete solutions. In this particular case, the idea of monads, already discovered and heavily used by the functional programming community to deal with side effects, turned out to be the perfect match to virtualize the variety, volume, and velocity dimensions of programming against collections.

Initially, LINQ—the incarnation of monads in .NET—targeted enumerable, or pull-based, in-memory (LINQ to Objects, LINQ to XML) and remote collections (LINQ to SQL, Entity Framework) with elements of various shapes (object, documents, rows). In the case of pull-based collections, the consumer synchronously requests subsequent items from a local or remote collection, and is blocked while the producer is busy generating the next value. Using the query comprehension syntax in C#, VB, and F#, developers can write declarative and compositional "queries" against enumerable collections, which can either be in-memory, such as an array or a list, or remote, such as a relational database.

Now you may ask yourself what this all has to do with orchestrating and coordinating event-based and asynchronous programs. That is where another mathematical trick comes in. By dualizing the IEnumerable/IEnumerator interfaces for synchronous collections, we get two interfaces, IObservable and IObserver, which characterize push-based, or observable streams. In this case, producers asynchronously notify consumers whenever a new item becomes available in the collection, and the consumer is unblocked to do whatever other work it needs to do between notifications.

The beauty of this trick is that we can use LINQ query comprehensions on either kind of collection. We can filter, transform, group, and join both synchronous, pull-based enumerable collections, as well as asynchronous, push-based observable collections, using exactly the same mechanism. In fact, we can even convert between the two, removing concurrency in one direction, and adding concurrency in the other. That brings us to the idea of schedulers, but you should read about those in the book.

In this book, Jesse and Paul do an amazing job explaining LINQ and the LINQ background of Rx, its relationship to standard design patterns, such as Subject/Observer, and peeking into the fundamental concepts of Rx, such as the IObservable/IObserver interfaces and schedulers. This book provides a wealth of information about applying Rx to real-world problems, such as wrapping .NET events and APM, testing asynchronous programs, using Rx.js in combination with various standard JavaScript libraries, and of course, simplifying the "Model-View-ViewModel Pattern" (note the careful hyphenation).

What I like in particular is that all concepts are clearly illustrated by artfully crafted code examples provided at precisely the right level of abstraction, addressing the topic at hand without leading the reader astray with unnecessary details. Developers can pick up this book and be up and running with Rx in minutes—no scary math required at all. But, you can be assured that everything you learn in this delightful book is built on strong foundations.

— Erik Meijer
Partner-Architect, Microsoft

Introduction

Right now, we as programmers are at an impasse—a transition period between the well-understood world of imperative programming, and a world that is increasingly at odds with this model. In the '80s, everything was simple: one machine, one thread, no network.

CPUs are now scaling horizontally, adding more and more cores, instead of scaling the CPU speeds. Network and disk performance are now increasingly requiring asynchronous I/O in order to build high-performance solutions.

Our tools to write asynchronous code, however, haven't kept up with the rest of the world—threads, locks, and events are the assembly language of asynchronous programming. They are straightforward to understand, but as the complexity of the application becomes larger, it becomes extremely difficult to determine if a given block of code will run correctly with respect to the rest of the application.

We need a way to retain the asynchronous nature of modern applications, while also retaining the deterministic nature of traditional imperative programming.

A compelling solution to these problems is functional reactive programming. This term sounds academic and overwhelming, however, if you've ever written a formula in Excel, good news—you've already done functional reactive programming.

Let's go through that title piece by piece, starting with the word "functional." Many people have a definition of functional programming, often from their college computer science course that covered Scheme or Common Lisp, usually something along the lines of, "it means never using variables."

Instead, think of FP as a *mindset*: "How is the output of my program *related* to my input, and how can I describe that relation in code?" In this book, we'll examine Language Integrated Queries (LINQ), a powerful technology that allows us to describe a result list based on an input list and a set of transformations on that input.

The disadvantage to LINQ is that this technology only works on lists, not data that is incoming or changing. However, there is a type of list that is hiding in plain sight, only disguised—events. Consider the KeyUp event: for every key that the user presses, an object representing the key will be generated—"H"–"e"–"l"–"l"–"o".

What if we thought of an event as a list? What if we could apply everything we know about lists to events, like how to filter them or create new lists based on existing lists?

Reactive Extensions allow you to treat asynchronous sources of information, such as events, and reason about them in the same way that you currently can reason about lists. This means, that once you become proficient with using LINQ to write single-threaded programs, you can apply your knowledge and write similar programs that are completely asynchronous.

A warning: this book is not easy to understand on first grasp. Rx will stretch your brain in ways it's not used to stretching, and you, like the authors, will almost certainly hit a learning curve.
The results, however, are most definitely worth it, as programs that would be incredibly difficult to write correctly otherwise, are trivially easy to write and reason with Reactive Extensions.

CHAPTER 1

Introducing LINQ and Rx

In a sentence, LINQ is a powerful way to interact with and extract data from collections, while Rx is an extension of LINQ targeted at asynchronous collections; that is, collections that will be populated asynchronously, typically from web services or elsewhere in the cloud.

LINQ was introduced to C# programmers with C# 3.0. On the other hand, Reactive Extensions (Rx) is an emerging technology, born in Microsoft DevLabs and adopted as part of a standalone full product from Microsoft in 2011.

We believe that the use of LINQ will expand greatly in the next few years, especially in conjunction with Reactive Extensions. Rx extends many of the principles and features of LINQ to a wide set of platforms (including JavaScript) and will be a vital part of Windows Phone programming, where asynchronous events are the rule. As the data we deal with becomes more complex, LINQ will help bring order and clarity to .NET programming. Similarly, as more and more data is retrieved asynchronously from the cloud, and elsewhere, Rx will help you create simpler, cleaner, and more maintainable code.

In this chapter you will learn why LINQ and Rx are important and you'll examine some of the fundamental operators of both LINQ and Rx. This will provide a context for the coming chapters in which we'll dive deeper into both LINQ and Reactive Extensions.

■ **Note** Throughout this book we use Rx as shorthand for Reactive Extensions, just as we use LINQ for Language Integrated Query.

What LINQ Is

LINQ (**L**anguage **In**tegrated **Q**uery)—pronounced "link"—extends .NET to provide a way to query and transform collections, relational data, and XML documents. It provides a SQL-like syntax within C# for querying data, regardless of where that data originates.

LINQ also brings a much more *declarative* or *functional* approach to programming than previously available in .NET.

■ **Note** Functional programming is one approach to creating declarative code.

In declarative programming, the logic and requirements are expressed but the execution steps are not.

LINQ and Rx are functional aspects of the C# language, in that they focus more on what you are trying to accomplish than on the steps required to get to that goal. Most programmers find that the functional approach makes it cleaner and easier to maintain code than the more traditional, imperative style of programming.

With the imperative approach, a program consists of a series of steps—often steps within loops—that detail what to do at any given moment. In declarative programming, we're able to express what is required at a higher level of abstraction.

It is the difference between saying on the one hand (imperative): "Open this collection, take each person out, test to see if the person is a manager, if the person is a manager—increase the salary by 20 percent, and put the person back into the collection;" and, on the other hand saying (declarative): "Raise the salary of all the managers by 20 percent."

The imperative style tells you how to accomplish the goal; the declarative style tells you what the goal is. Using concrete programming examples of Rx and LINQ, this book will demonstrate differences that are more declarative than imperative.

What Rx Is

Reactive Extensions provide you with a new way to orchestrate and integrate asynchronous events, such as coordinating multiple streams as they arrive asynchronously from the cloud. With Rx you can "flatten" these streams into a single method, enormously simplifying your code. For example, the classic async pattern in .NET is to initiate each call with a BeginXXX method and end it with an EndXXX method, also known as the Begin/End pattern. If you make more than a few simultaneous asynchronous calls, following the thread of control becomes impossible very quickly. But with Rx, the Begin/End pattern is collapsed into a single method, making the code much cleaner and easier to follow.

Reactive Extensions have been described as a library for *composing asynchronous* and *event-based* programs using *observable collections*.[1]

Let's take a closer look at each of these attributes.

- ***Composing***: One of the primary goals of Reactive Extensions is to make the work of combining asynchronous operations significantly easier. To do this, the data flow must be clarified and the code consolidated so that it is less spread out throughout your application.
- ***Asynchronous***: While not everything you'll do in Rx is asynchronous, the async code you write will be simpler, easier to understand, and thus far easier to maintain.

[1] MSDN Data Developer Center, "Reactive Extensions," `http://msdn.microsoft.com/en-us/data/gg577609`.